Embedded Real-Time Control, Alberto Morato

Laboratory 0, GPIOs and Interrupts

Last update: March 19, 2024

1 Introduction

This experience is focused on the use of GPIOs and interrupts.
The TurtleBot has several buttons, switches, and LEDs connected to the STM32F767 GPIOs.

1.1 Button

One of the buttons is connected to the pin `PF8 (Pin 8, Port F)` and it is labeled as `GPIO_EXTI8_USER_BUT1_IRQ`. Normally the GPIO line is forced to ground by using the series of pull-down resistors `R6 and R5`. The capacitor `C1` combined with `R6 and R5`, form a low-pass filter (de-bounce circuit).

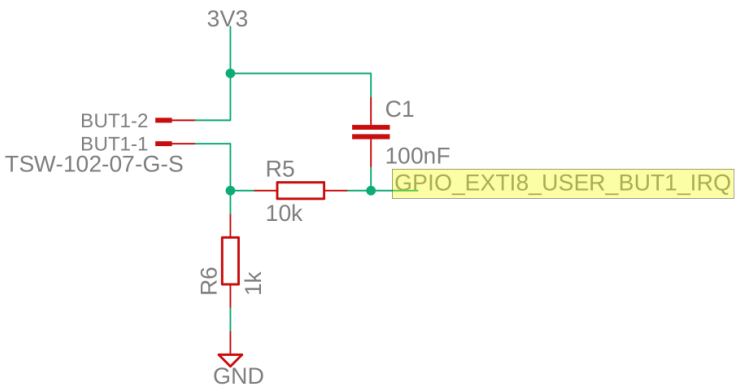


Figure 1: Button

1.2 LED

One of the LEDs is connected to the pin `PE5 (Pin 5, Port E)` and it is labeled as `TIM9_CH1_USER_LED1`. The LED is connected directly to the GPIO using a current limiting resistor `R1`.

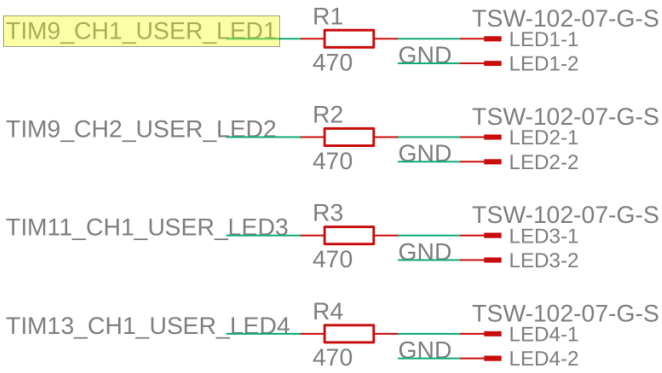


Figure 2: LED

2 STM32CubeIDE and STM32CubeMX Overview

STM32CubeIDE is an integrated development environment (IDE) provided by STMicroelectronics for STM32 microcontroller development. It's based on the popular Eclipse IDE platform and is tailored specifically for STM32 microcontroller projects. Here's an overview of its features and functionalities:

1. **Integrated Development Environment (IDE)**:
   - STM32CubeIDE provides a comprehensive development environment for writing, compiling, debugging, and flashing STM32 microcontroller firmware.

- It offers a familiar and user-friendly interface, as it is based on Eclipse IDE, which is widely used in the software development community.

2. **Code Generation and Configuration**:
   - STM32CubeIDE integrates seamlessly with STM32CubeMX, a graphical configuration tool for STM32 microcontrollers.
   - Developers can configure various aspects of their microcontroller projects using STM32CubeMX and then import the generated code directly into STM32CubeIDE.
   - This integration streamlines the process of configuring peripherals, setting up the clock tree, and generating initialization code for STM32 microcontroller projects.

STM32CubeMX is a tool integrated with STM32CubeIDE that allows users to configure and initialize the STM32 microcontroller. Among its features, we can mention:

- **Graphical Configuration Interface:** STM32CubeMX provides a user-friendly graphical interface where you can select the STM32 microcontroller model you're using and configure various parameters such as clock settings, GPIO pins, peripherals (e.g., UART, SPI, I2C), interrupt priorities, and middleware components.
- **Pinout Configuration:** One of the essential features of STM32CubeMX is the pinout configuration tool. It allows you to assign functions to specific GPIO pins, including configuring pins for alternate functions (e.g., UART, SPI, PWM). You can visually see the pinout of the microcontroller and make assignments according to your project requirements.
- **Code Generation:** After configuring the microcontroller and peripherals, STM32CubeMX generates initialization code in C language based on your settings. This code initializes the microcontroller's peripherals and sets up the system according to your configuration. The generated code includes startup code, system initialization, peripheral initialization, and interrupt handlers.

3 Create a New Project

1. Open STM32CubeIDE
2. File -> New -> STM32 Project from an existing STM32CubeMX Configuration File

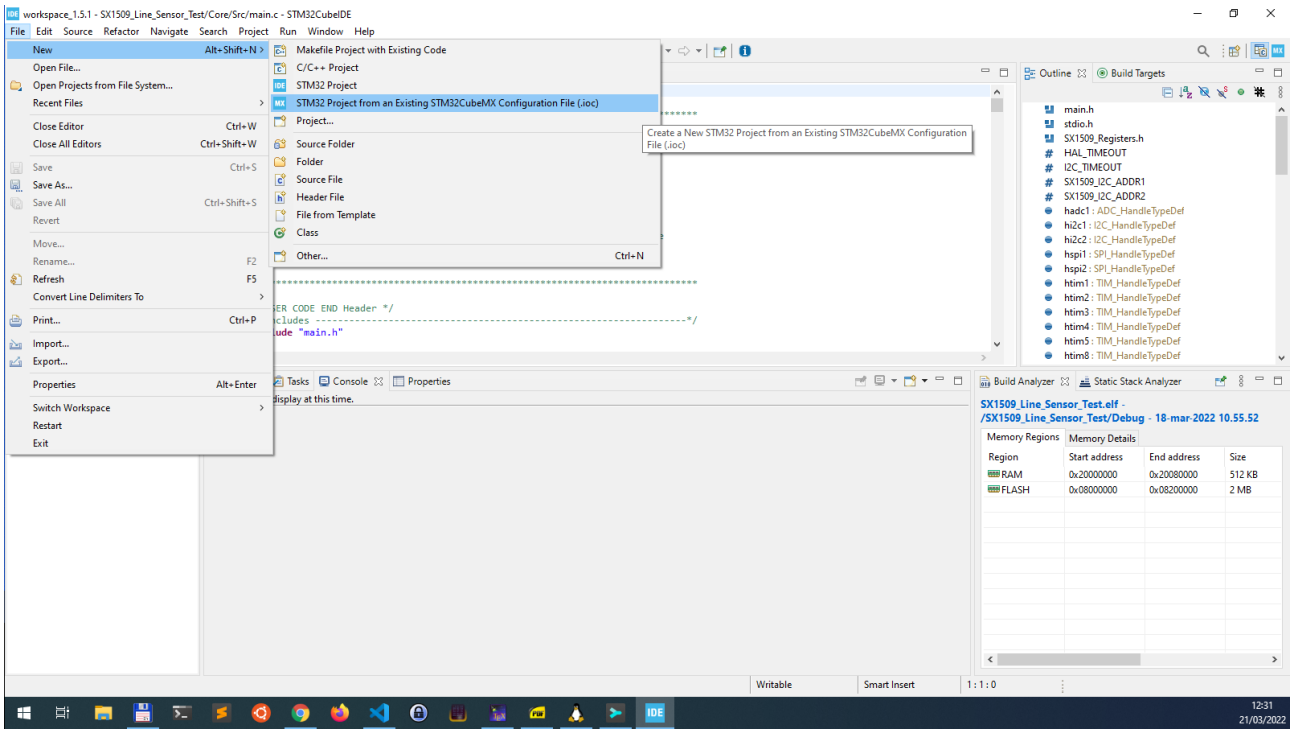


Figure 3: Step 1

3. Select the provided `*.ioc` file and give a name to the project. Select `C` as the target language. Click `Next`.

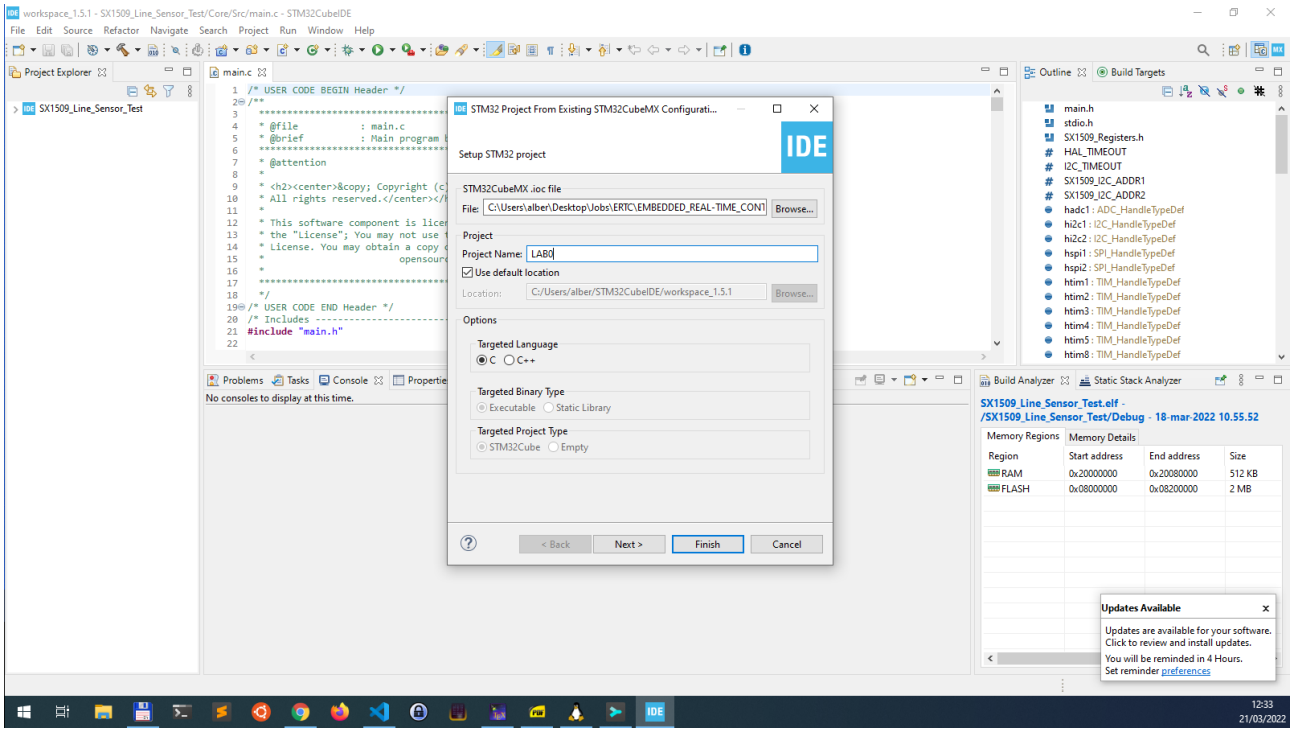


Figure 4: Step 2

4. Select `Copy only the necessary library files` and click `Finish`.

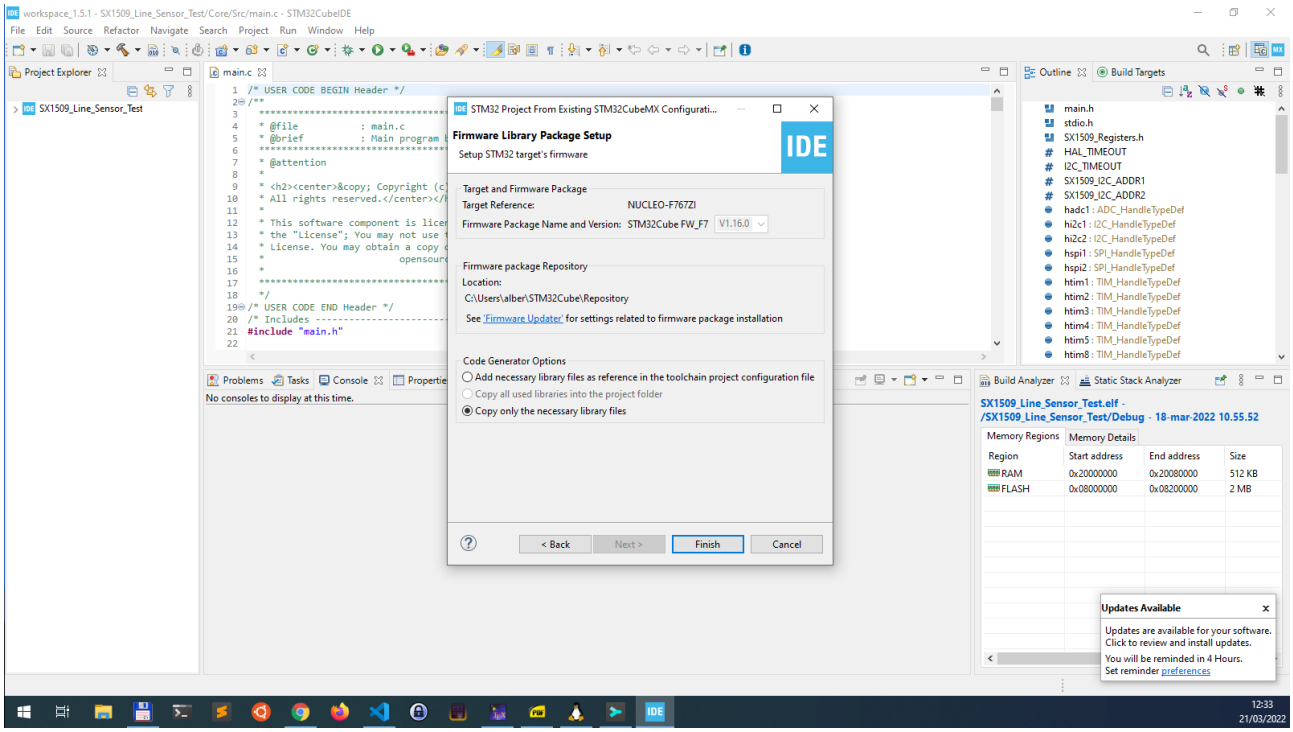


Figure 5: Step 3

5. Click the gear icon on the toolbar. In this way, the tool will generate all the necessary HAL functions and helpers.

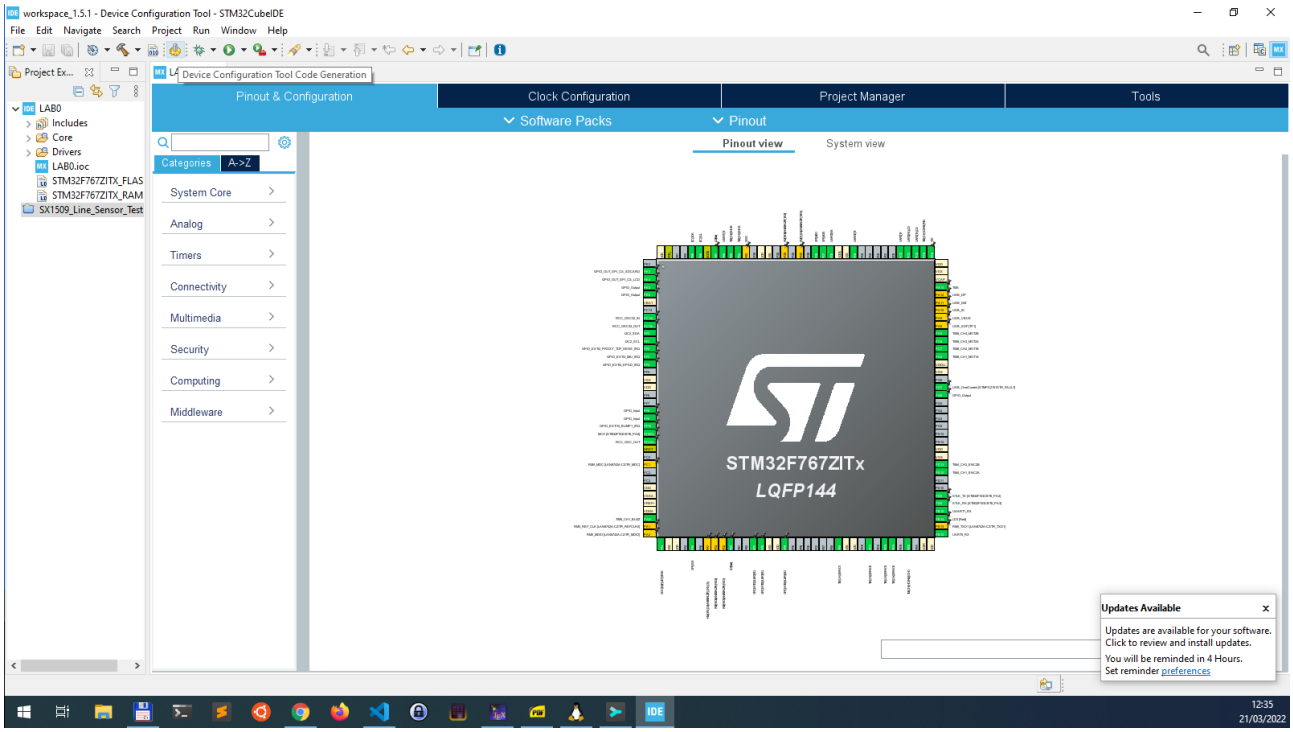


Figure 6: Step 4

4 Structure of the `main.c` File

This main file is typically generated by STM32CubeMX when you configure your STM32 microcontroller project. It contains several sections. For example in the first part is possible to include libraries, define macros, custom types, user defined global variables, and user defined functions.

The `int main(void)` function is the entry point of the program. It is the first function that is called when the program starts. Inside this function, there is an endless while loop where the user can put the code that needs to be executed forever.

Inside the generated code there are some comments that help the user to understand where to put the code. In particular, the `USER CODE BEGIN` and `USER CODE END` comments are strategically placed to allow developers to insert their own code within the generated code without worrying about it being overwritten when the code is regenerated. This separation makes it easier to maintain and update the codebase, especially when using code generation tools like STM32CubeMX.

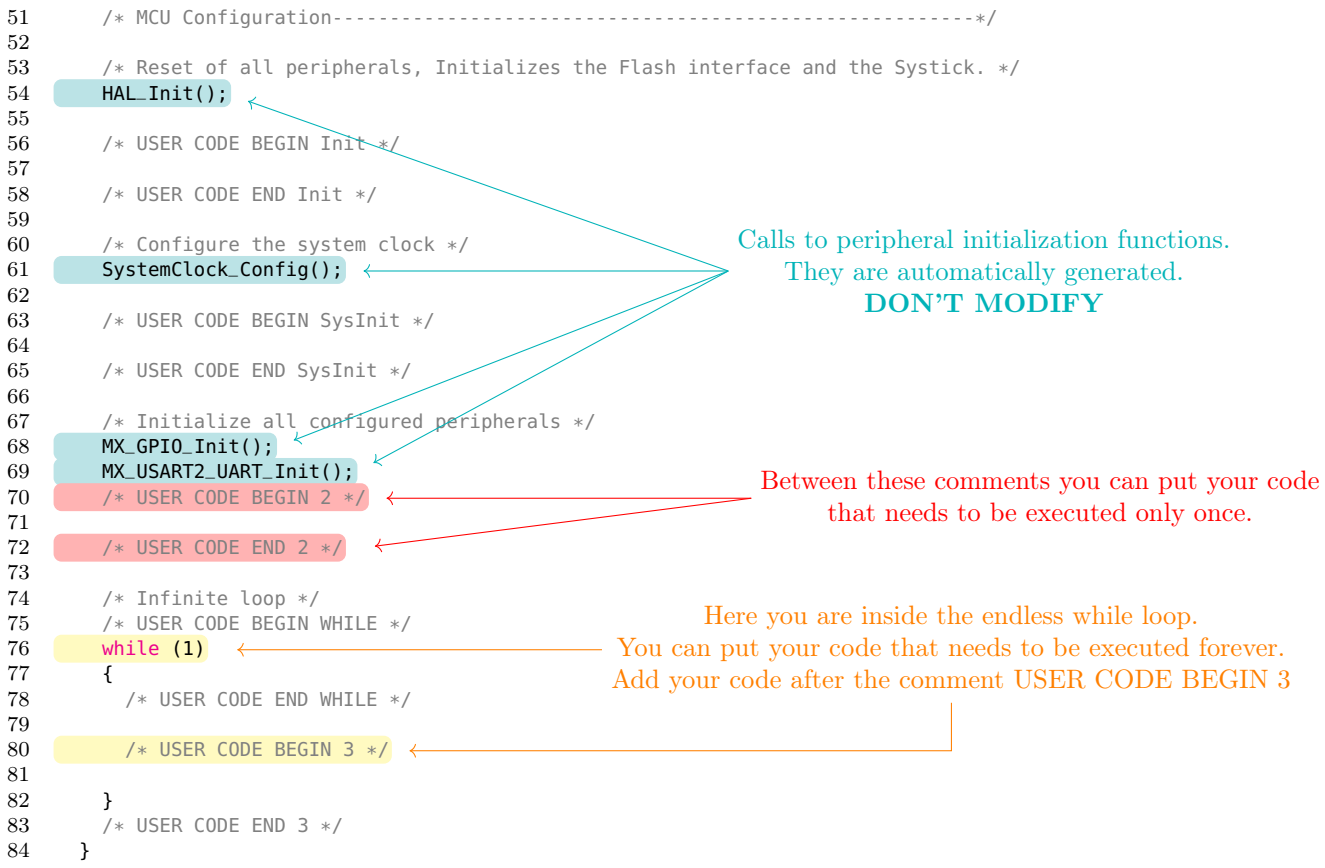
```c
#include "main.h"

/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */

/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */

/* USER CODE END PTD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */
/* USER CODE END PD */

/* Private macro -------------------------------------------------------------*/
/* USER CODE BEGIN PM */

/* USER CODE END PM */

/* Private variables ---------------------------------------------------------*/
UART_HandleTypeDef huart2;

/* USER CODE BEGIN PV */

/* USER CODE END PV */

/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART2_UART_Init(void);
/* USER CODE BEGIN PFP */

/* USER CODE END PFP */

/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */

/* USER CODE END 0 */

/**
  * @brief  The application entry point.
  * @retval int
  */
int main(void)
{
  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */
  ...
```

Here you are at the beginning of the `c` file.
Here, usually are included the libraries (`#include`),
defined macros (`#define`) or custom types (`typedef ...`).
You can define them between these two comments.

Definition of variables and function used by the HAL.
They are automatically generated.
DON'T MODIFY

Here you can define your global variables and functions.
They will be accessible from all the file.
Add your code after the comment USER CODE BEGIN 0.

This is the entry point of your program.



5 HAL Functions

Following are useful HAL functions for this lab. More detailed information [1].

1. `HAL_GPIO_WritePin(GPIOX, GPIO_PIN_Y, state)`: Sets the state of a GPIO pin given the PORT, pin number, and state. The `state` parameter can be either `GPIO_PIN_RESET` or `GPIO_PIN_SET`. Alternatively, you can use **0** or **1**.
2. `HAL_GPIO_ReadPin(GPIOX, GPIO_PIN_Y)`: Reads the state of a GPIO pin given the PORT and pin number.
3. `HAL_GPIO_TogglePin(GPIOX, GPIO_PIN_Y)`: Toggles the state of a GPIO pin given the PORT and pin number.
4. `HAL_Delay(uint32_t Delay)`: Delays execution for a given number of milliseconds.

In the examples above **X** should be replaced with the capital letter of the port (A, B, C, ...) where the GPIO is located. **Y** should be replaced with the number of the GPIO pin within the port. For example, to set the state of the pin PE5, you should use `HAL_GPIO_WritePin(GPIOE, GPIO_PIN_5, GPIO_PIN_SET)`.

6 Building the project and flashing the μC

Once you have written your code, you can build the project by clicking on the hammer icon in the toolbar. This will compile the code and generate the necessary binary files for flashing the microcontroller.

If you have any errors or warnings, you can view them in the "Problems" tab at the bottom of the screen. This will show you any issues that need to be addressed before you can successfully build and flash the project.

To flash the microcontroller, you can click on the "play" icon in the toolbar. This will upload the binary files to the microcontroller and start the program.

Alternatively, you can use the "Debug" button to start a debugging session, which allows you to step through the code, set breakpoints, and inspect variables in real-time.

[1] https://www.st.com/resource/en/user_manual/dm00189702-description-of-stm32f7-hal-and-lowlayer-drivers-stmicroelectronics.pdf

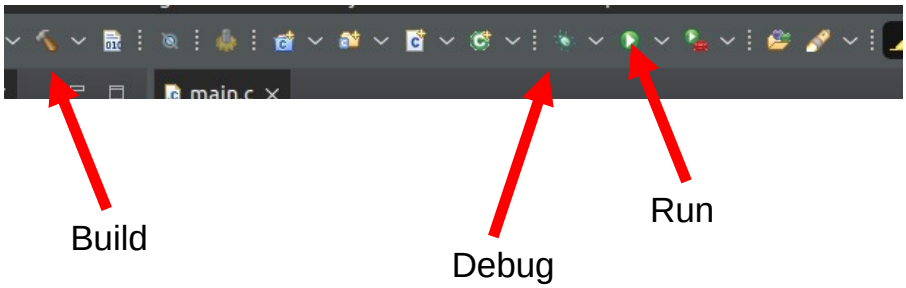


Figure 7: Toolbar

7 Exercises

7.1 Exercise 1

Run this sequence **once**: PE5

1. Turn the LED **ON**.
2. Wait for 1[s] (you can use the function `HAL_Delay(uint32_t delay)` to wait for delay milliseconds).
3. Turn the LED **OFF**.

7.2 Exercise 2 PE5

Run this sequence **forever**:

1. Read the state of the button. PF8
2. If it is pressed, turn **ON** the LED, **OFF** otherwise.
3. To simulate the time required to run other portions of the code, try different delays between two readings of the button state; Which is the effect?

7.3 Exercise 3

1. Configure the GPIO associated with the button as an interrupt source; you can do this by opening the *.ioc file from the "Project Explorer" bar; You will have a screen similar to this.

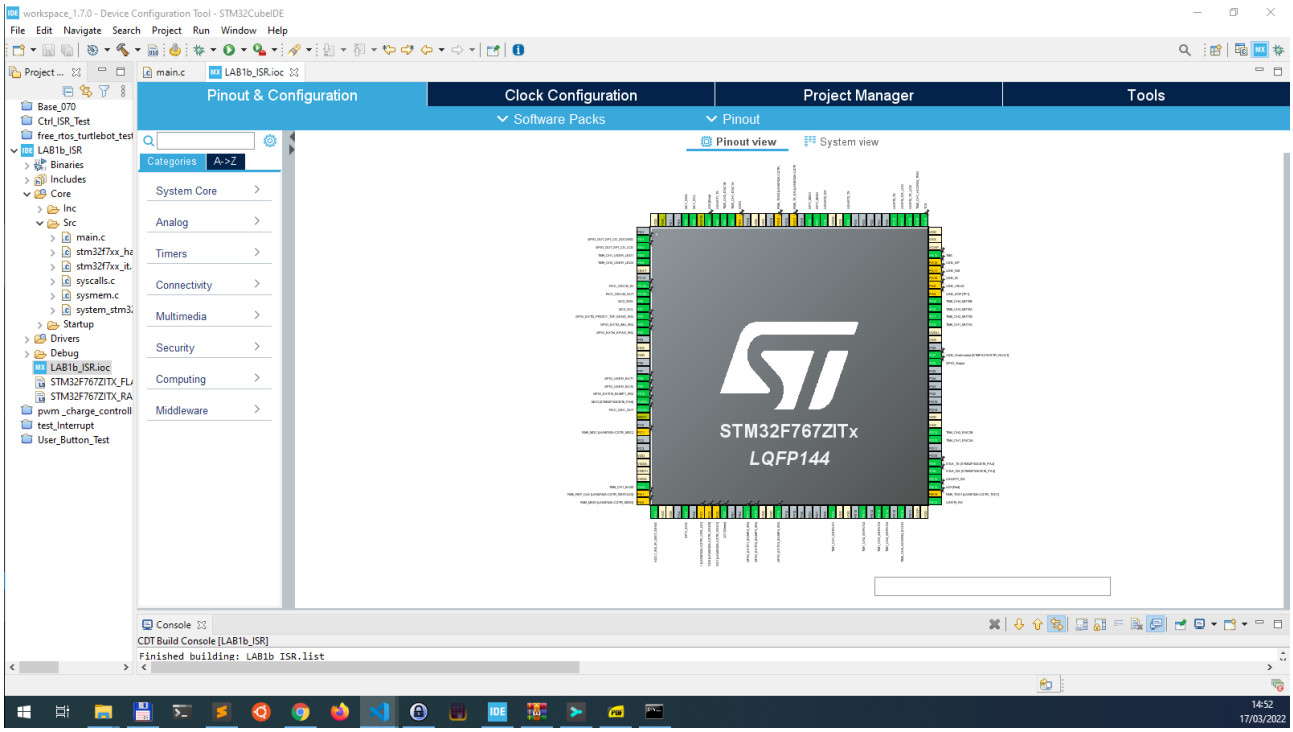


Figure 8: Step 1

1. Search for pin PF8; Click on that and then click `GPIO_EXTI8`. In this way, the GPIO is connected to the interrupt line 8.

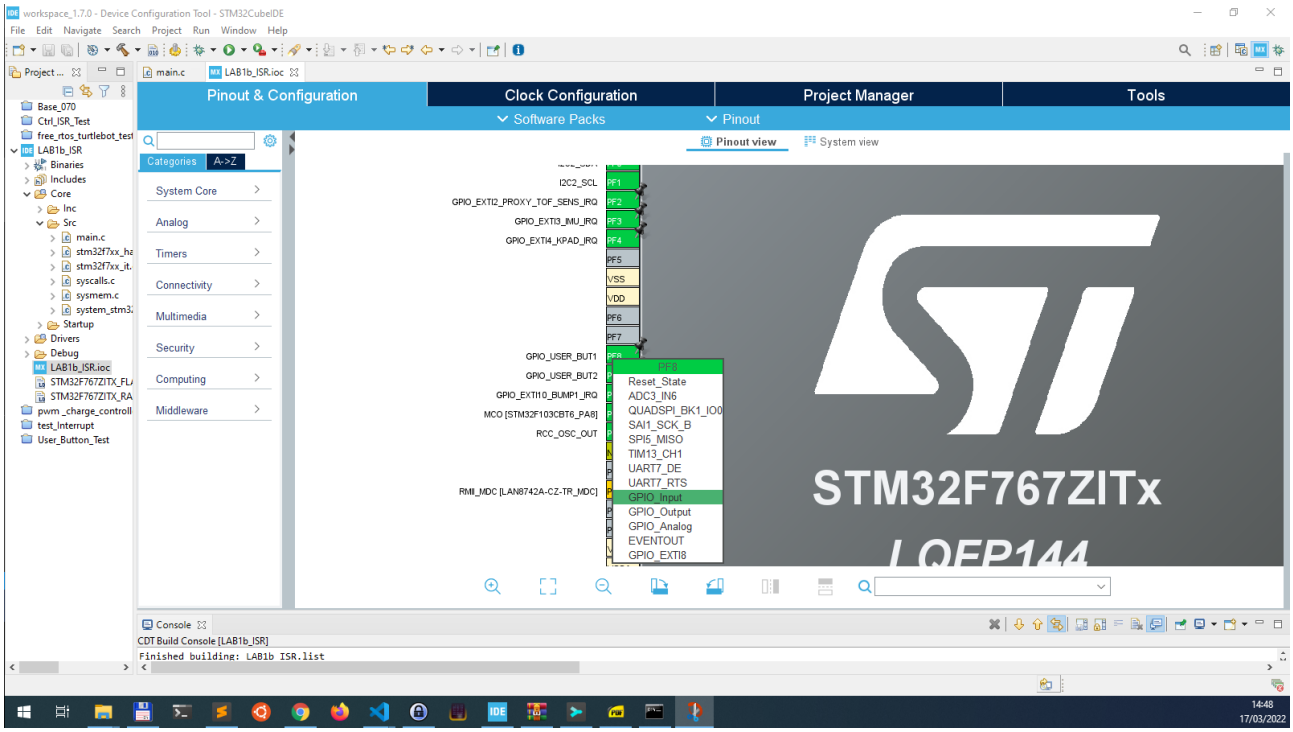


Figure 9: Step 2

1. Click on `System Core > GPIO`. Search for pin PF8 and click on that. Here you can set the trigger mode for the interrupt and the pull/push up/down configuration. Select `External Interrupt Mode with Rising edge trigger detection` and `No pull-up and no pull-down`.

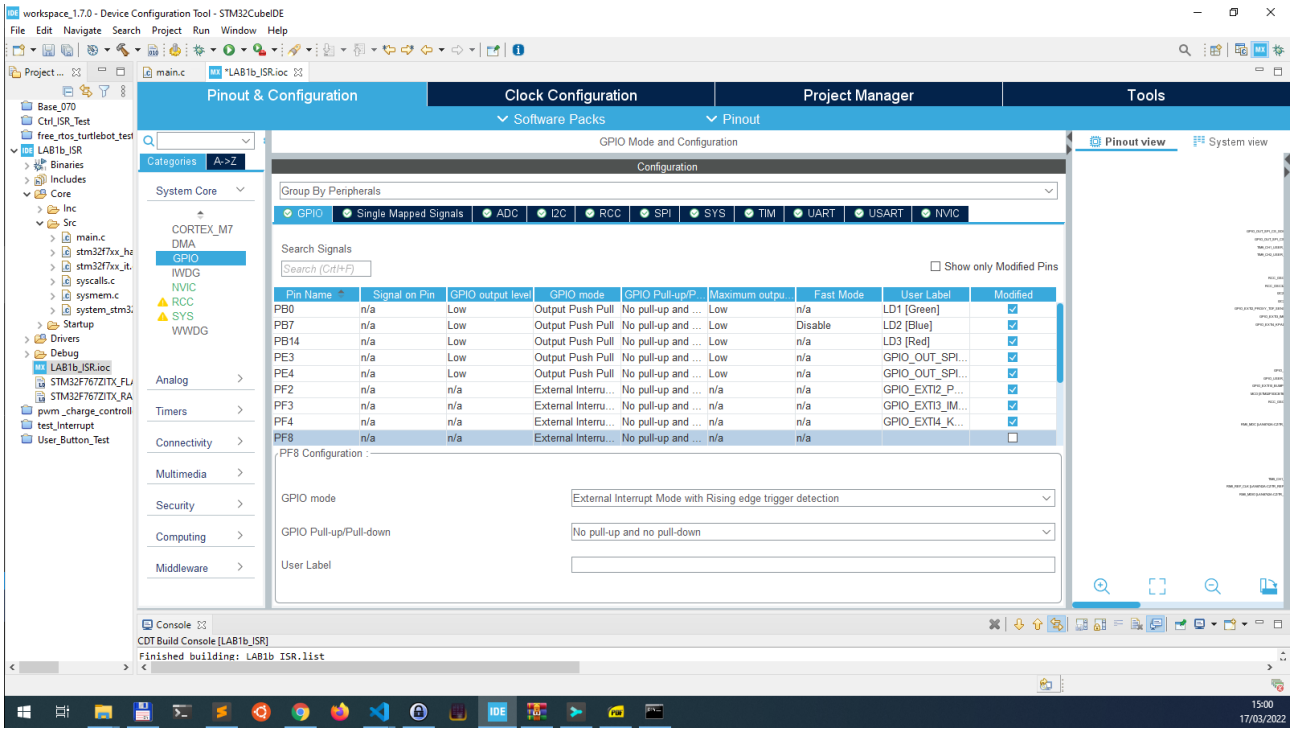


Figure 10: Step 3

1. Click on `System Core > NVIC`. Search for `EXTI line[9:5]` and enable it. In this way, you have enabled all the interrupts for line 5 to 9. Here you can also change the priority. priority 0 highest or lowest

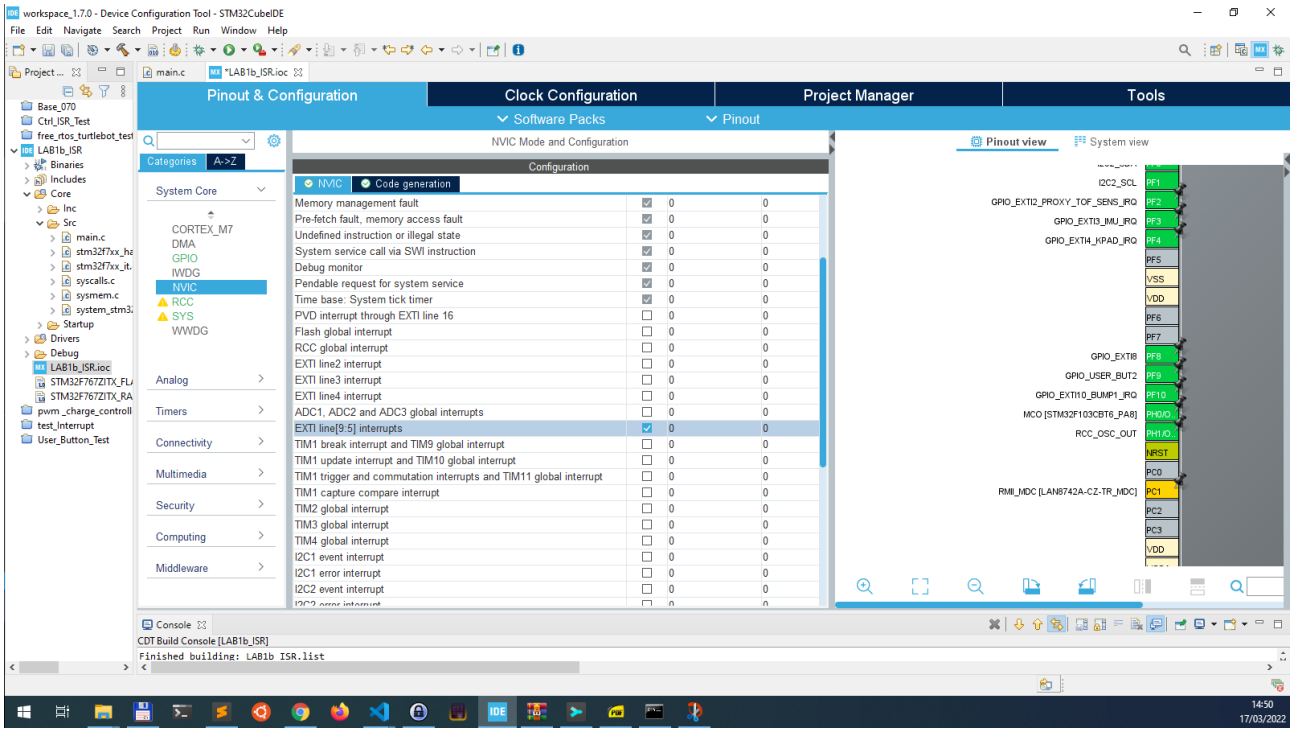


Figure 11: Step 4

1. Save (CTRL-S).
2. Remove the code you wrote inside `main.c` for exercises 1 and 2. Make a backup of the `main.c` file in case you need it in the future.
3. Inside the `main.c` file, define a function called `void HAL_GPIO_EXTI_Callback(uint16_t pin)`; this function is automatically called when the interrupt occurs.

4. Toggle the state of the LED every time the button changes state from “not pressed” to “pressed”.

7.4 Exercise 4

Assuming that the button is configured as an interrupt source, as in Exercise 3:

1. Apply the sequence (LED ON -> wait -> LED OFF) from Exercise 1, but instead of doing that once, repeat the sequence forever.
2. Every time the button changes state from “not pressed” to “pressed”, decrease the wait time by 100[ms]. If the wait time reaches 0, reset it to 1[s].

7.5 Bonus

The TurtleBot has another LED and button connected respectively to `PE6` and `PF9`. Try to use also these ones. Have fun!